**Capítulo TREZE**

**Iterando e Filtrando Coleções**

**Objetivos do Exame**

- Streams e Filtros de Coleções.
- Iterar usando métodos forEach de Streams e List.
- Filtrar uma coleção usando expressões lambda.

---

**Iteração**

Geralmente, quando você tem uma lista, você quer iterar sobre seus elementos. Uma maneira comum é usar um bloco for.

Ou com um índice:

```java
List<String> words = ...
for(int i = 0; i < words.size(); i++) {
    System.out.println(words.get(i));
}
```

Ou com um iterador:

```java
List<String> words = ...
for(Iterator<String> it = words.iterator(); it.hasNext();) {
    System.out.println(it.next());
}
```

Ou com o chamado loop *for-each*:

```java
List<String> words = ...
for(String w : words) {
    System.out.println(w);
}
```

Além de parecer feio, os dois primeiros adicionam pontos onde um erro pode acontecer (as variáveis de índice e iterador). A forma recomendada é usar o *for-each* sempre que possível.

Portanto, aproveitando as interfaces funcionais, o Java 8 adiciona outra opção para iterar listas baseada no *for-each*, o método foreach:

```java
default void forEach(Consumer<? super T> action)
```

Como este método é definido na interface Iterable, ele não está disponível apenas para listas, mas para todas as implementações desta interface, como Queues, Sets, Deques e até algumas exceções relacionadas a SQL, como SQLException.

Observe também que este é um método default, o que significa que há uma implementação padrão que as classes que implementam podem sobrescrever (e muitas fazem, principalmente para lidar com modificações concorrentes).

Esta é a implementação padrão:

```java
for (T t : this) {
    action.accept(t);
}
```

Basicamente, é um loop *for-each* usando os novos recursos funcionais do Java 8.

Para usá-lo, podemos começar com uma classe anônima:

```java
List<String> words = ...
words.forEach(new Consumer<String>() {
    public void accept(String t) {
        System.out.println(t);
    }
});
```

Lembre-se de que a interface Consumer representa uma operação que recebe um parâmetro, mas não retorna nenhum resultado.

Essa classe anônima pode ser transformada em uma expressão lambda:

```java
words.forEach(t -> System.out.println(t));
```

Ou, neste exemplo específico, uma referência de método:

```java
words.forEach(System.out::println);
```

Lembre-se das regras sobre o uso de variáveis final ou efetivamente final dentro de classes anônimas ou expressões lambda. Código como o seguinte não é válido:

```java
int max = 0;
words.forEach(t -> {
    // A linha a seguir não compila, você não pode modificar max
    max = Math.max(max, t.length());
    System.out.println(t);
});
```

Se você quiser fazer coisas assim (obter o comprimento máximo de todas as strings em uma lista), é melhor usar *streams* para iterar sobre a coleção e aplicar outras operações (para esse exemplo específico, veremos como calcular o comprimento máximo com o método reduce em um capítulo posterior).

A interface Stream fornece um método forEach correspondente:

```java
void forEach(Consumer<? super T> action)
```

Como este método não retorna um stream, ele representa uma operação terminal.

Usá-lo não é diferente da versão Iterable:

```java
Stream<String> words = ...
// Como classe anônima
words.forEach(new Consumer<String>() {
    public void accept(String t) {
        System.out.println(t);
    }
});
// Como expressão lambda
words.forEach(t -> System.out.println(t));
// Como referência de método
words.forEach(System.out::println);
```

Claro, a vantagem de usar streams é que você pode encadear operações, por exemplo:

```java
words.sorted()
     .limit(2)
     .forEach(System.out::println);
```

Como é uma operação terminal, você não pode fazer coisas como:

```java
words.forEach(t -> System.out.println(t.length()));
words.forEach(System.out::println);
```

Se quiser fazer algo assim, crie um novo stream a cada vez:

```java
Stream.of(wordList).forEach(t -> System.out.println(t.length()));
Stream.of(wordList).forEach(System.out::println);
```

Ou encapsule o código dentro de uma única lambda:

```java
Consumer<String> print = t -> {
    System.out.println(t.length());
    System.out.println(t);
};
words.forEach(print);
```

Você não pode usar return, break ou continue para terminar uma iteração. break e continue gerarão um erro de compilação, pois não podem ser usados fora de um loop, e return não faz sentido quando vemos que o método foreach é basicamente implementado assim:

```java
for (T t : this) {
    // Dentro do accept, return não tem efeito
    action.accept(t);
}
```

Como nota (já que não é coberto no exame), o Java 8 também adicionou um método forEach à interface Map. No entanto, como um mapa tem uma chave e um valor, este novo método usa um BiConsumer:

```java
default void forEach(BiConsumer<? super K, ? super V> action)
```

Com uma implementação padrão equivalente a:

```java
for (Map.Entry<K, V> entry : map.entrySet()) {
    action.accept(entry.getKey(), entry.getValue());
}
```

---

**Filtragem**

Outro requisito comum é filtrar (ou remover) elementos de uma coleção que não correspondam a uma condição específica.

Normalmente você faz isso copiando os elementos correspondentes para outra coleção:

```java
List<String> words = ...
List<String> nonEmptyWords = new ArrayList<String>();
for (String w : words) {
    if (w != null && !w.isEmpty()) {
        nonEmptyWords.add(w);
    }
}
```

Ou removendo os elementos que não correspondem diretamente da coleção usando um iterador (somente se a coleção suportar remoção):

```java
List<String> words = new ArrayList<String>();
// ... (adiciona algumas strings)
for (Iterator<String> it = words.iterator(); it.hasNext();) {
    String w = it.next();
    if (w == null || w.isEmpty()) {
        it.remove();
    }
}
```

Agora, no Java 8, há um novo método na interface Collection:

```java
default boolean removeIf(Predicate<? super E> filter)
```

Que remove todos os elementos da coleção que satisfaçam o predicado fornecido (a implementação padrão usa a versão com iterador).

Isso torna o código mais simples, usando expressões lambda ou referências de método:

```java
// Usando uma classe anônima
words.removeIf(new Predicate<String>() {
    public boolean test(String t) {
        return t == null || t.isEmpty();
    }
});
// Usando uma expressão lambda
words.removeIf(t -> t == null || t.isEmpty());
```

Para o caso em que você copia os elementos correspondentes para outra coleção, você tem o método filter da interface Stream:

```java
Stream<T> filter(Predicate<? super T> predicate)
```

Que retorna um novo stream contendo os elementos que satisfazem o predicado fornecido.

Como este método retorna um stream, ele representa uma operação intermediária, o que basicamente significa que você pode encadear qualquer número de filtros ou outras operações intermediárias:

```java
List<String> words = Arrays.asList("hello", null, "");
words.stream()
     .filter(t -> t != null) // ["hello", ""]
     .filter(t -> !t.isEmpty()) // ["hello"]
     .forEach(System.out::println);
```

Claro, o resultado da execução desse código é:

```nginx
hello
```

Você também pode criar um método (ou usar um já existente) em alguma classe para fazer isso com uma referência de método, apenas por questão de clareza:

```java
class StringUtils {
    public static boolean isNotNullOrEmpty(String s) {
        return s != null && !s.isEmpty();
    }
}

// ...

List<String> words = Arrays.asList("hello", null, "");

// Usando classe anônima
words.stream()
     .filter(new Predicate<String>() {
         public boolean test(String t) {
             return StringUtils.isNotNullOrEmpty(t);
         }
     })
     .forEach(System.out::println);

// Usando expressão lambda
words.stream()
     .filter(t -> StringUtils.isNotNullOrEmpty(t))
     .forEach(System.out::println);

// Usando referência de método
words.stream()
     .filter(StringUtils::isNotNullOrEmpty)
     .forEach(System.out::println);
```

A interface Stream também possui o método distinct para filtrar elementos duplicados, de acordo com o método Object.equals(Object):

```java
Stream<T> distinct()
```

Novamente, como ele retorna um novo stream, essa é uma operação intermediária. Como ele precisa conhecer os valores dos elementos para identificar duplicatas, essa operação também é com estado (*stateful*).

Aqui está um exemplo:

```java
List<String> words = Arrays.asList("hello", null, "hello");
words.stream()
     .filter(t -> t != null) // ["hello", "hello"]
     .distinct() // ["hello"]
     .forEach(System.out::println);
```

O resultado é:

```nginx
hello
```

---

**Pontos-Chave**

O Java 8 adiciona o seguinte método à interface Iterable como outra opção para iterar sobre implementações dessa interface (como listas):

```java
default void forEach(Consumer<? super T> action)
```

Por exemplo:

```java
List<String> words = Arrays.asList("hello", "world");
words.forEach(t -> System.out.println(t));
```

A interface Stream também possui esse método:

```java
void forEach(Consumer<? super T> action)
```

Essa é uma operação terminal. Aqui está um exemplo:

```java
Stream<String> words = Stream.of("hello", "world");
words.forEach(t -> System.out:println(t));
```

Claro, a vantagem de usar streams é que você pode encadear operações, por exemplo:

```java
words.sorted()
     .limit(2)
     .forEach(System.out::println);
```

Mas como se trata de uma operação terminal, você **não pode** fazer coisas como:

```java
words.forEach(t -> System.out.println(t.length()));
words.forEach(System.out::println);
```

Para filtragem, no lado das coleções, temos um novo método:

```java
default boolean removeIf(Predicate<? super E> filter)
```

Que remove todos os elementos da coleção que satisfaçam o predicado fornecido.

Na interface Stream, temos:

```java
Stream<T> filter(Predicate<? super T> predicate)
```

Que retorna um novo stream composto pelos elementos que satisfazem o predicado fornecido.

Como este método retorna um stream, ele representa uma operação intermediária, o que significa que você pode encadear qualquer número de filtros ou outras operações intermediárias:

```java
List<String> words = Arrays.asList("hello", null, "");
words.stream()
     .filter(t -> t != null) // ["hello", ""]
     .filter(t -> !t.isEmpty()) // ["hello"]
     .forEach(System.out::println);
```

A interface Stream também possui o método distinct para filtrar elementos duplicados, conforme o método Object.equals(Object):

```java
Stream<T> distinct()
```

Essa é uma operação intermediária e, como precisa conhecer os valores dos elementos para identificar os duplicados, essa operação também é com estado (*stateful*).

Aqui está um exemplo:

```java
List<String> words = Arrays.asList("hello", null, "hello");
words.stream()
     .filter(t -> t != null) // ["hello", "hello"]
     .distinct() // ["hello"]
     .forEach(System.out::println);
```

---

**Autoavaliação**

**1. Dado:**

```java
public class Question_13_1 {
    public static void main(String[] args) {
        List<Integer> l = Arrays.asList(1, 2, 3, 4, 5, 6);
        Stream.of(l)
              .forEach(i -> System.out.print(i - 1));
    }
}
```

Qual é o resultado?

A. 123456
B. 012345
C. 543210
D. Erro de compilação
E. Uma exceção é lançada

---

**2. Qual das seguintes afirmações é verdadeira?**

A. filter é uma operação terminal.
B. filter é uma operação com estado (*stateful*).
C. Stream.forEach recebe uma implementação da interface funcional Consumer como argumento.
D. Você pode encadear mais de uma operação forEach em um pipeline de stream.

---

**3. Dado:**


Qual é o resultado?

A. 246
B. 46
C. 1
D. 5
E. Erro de compilação

---

**4. Dado:**


Qual é o resultado?

A. 1
B. 0
C. Nada é impresso
D. Erro de compilação
E. Uma exceção é lançada